High Performance Biological Pairwise Sequence Alignment: FPGA versus GPU versus Cell BE versus GPP

Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2012, Article ID 752910, 15 pages
doi:10.1155/2012/752910

Review Article

High Performance Biological Pairwise Sequence Alignment: FPGA versus GPU versus Cell BE versus GPP

Khaled Benkrid,[1] Ali Akoglu,[2] Cheng Ling,[1] Yang Song,[2] Ying Liu,[1] and Xiang Tian[1]

[1] *Institute of Integrated Systems, School of Engineering, The University of Edinburgh, Kings Buildings, Mayfield Road, Edinburgh EH9 3JL, UK*
[2] *Electrical and Computer Engineering Department, The University of Arizona, Tucson, AZ 85721-0104, USA*

Correspondence should be addressed to Khaled Benkrid, k.benkrid@ed.ac.uk

Received 15 December 2011; Revised 13 February 2012; Accepted 17 February 2012

Academic Editor: Kentaro Sano

This paper explores the pros and cons of reconfigurable computing in the form of FPGAs for high performance efficient computing. In particular, the paper presents the results of a comparative study between three different acceleration technologies, namely, Field Programmable Gate Arrays (FPGAs), Graphics Processor Units (GPUs), and IBM's Cell Broadband Engine (Cell BE), in the design and implementation of the widely-used Smith-Waterman pairwise sequence alignment algorithm, with general purpose processors as a base reference implementation. Comparison criteria include speed, energy consumption, and purchase and development costs. The study shows that FPGAs largely outperform all other implementation platforms on performance per watt criterion and perform better than all other platforms on performance per dollar criterion, although by a much smaller margin. Cell BE and GPU come second and third, respectively, on both performance per watt and performance per dollar criteria. In general, in order to outperform other technologies on performance per dollar criterion (using currently available hardware and development tools), FPGAs need to achieve at least two orders of magnitude speed-up compared to general-purpose processors and one order of magnitude speed-up compared to domain-specific technologies such as GPUs.

1. Introduction

Since it was first announced in 1965, Moore's law has stood up the test of time, providing exponential increases in computing power for science and engineering problems over time. However, while this law was largely followed through increases in transistor integration levels and clock frequencies, this is no longer possible as power consumption and heat dissipation are becoming major hurdles in the face of further clock frequency increases, the so-called frequency or power wall problem.

In order to keep Moore's law going, general-purpose processor manufacturers, for example, Intel and AMD, are now relying on multicore chip technology in which multiple cores run simultaneously on the same chip at capped clock frequencies to limit power consumption. While this has the potential to provide considerable speed-up for science and engineering applications, it is also creating a semantic gap between applications, traditionally written in sequential code, and hardware, as multicore technologies need to be programmed in parallel to take advantage of their performance potential. This problem is however also opening a window of opportunity for hitherto niche parallel computer technologies such as Field Programmable Gate Arrays (FPGAs) and Graphics Processor Units (GPUs) since the problem of parallel programming has to be tackled for general-purpose processors anyway.

This paper presents a comparative study between three different acceleration technologies, namely, Field Programmable Gate Arrays (FPGAs), Graphics Processor Units (GPUs), and IBM's Cell Broadband Engine (Cell BE), in the design and implementation of the widely-used Smith-Waterman pairwise sequence alignment algorithm, with general purpose processors as a base reference implementation. Comparison criteria include the speed of the resulting implementation, its energy consumption, as well as purchase

and development costs. Note that the aim of this paper is not to present the best implementation (from a speed point of view) on the four architectures but to perform a fair comparison of all four technologies in terms of speed, energy consumption, and development time and cost. We thus chose not to use the results of the best implementations reported in the literature, but instead to perform our own experiments using a set of Ph.D. students with relatively equal experience on each platform and measure the speed, development time, cost and energy consumption of each resulting implementation.

The rest of this paper is organized as follows. The following section will first present background on the Smith-Waterman algorithm, together with an overview of the target implementation platforms, namely, Xilinx Virtex-4 FPGAs, NVIDIA GeForce 8800GTX GPU, IBM's Cell BE processor and finally the Pentium 4 Prescott processor. Sections 3, 4, 5, and 6 will then report our design and implementation of the Smith-Waterman algorithm on each of the above platforms, in turn. After that, comparative implementation results on all platforms are presented in Section 7 before final conclusions are drawn.

2. Background

Pairwise biological sequence alignment is a basic operation in the field of bioinformatics and computational biology with a wide range of applications in disease diagnosis, drug engineering, biomaterial engineering, and genetic engineering of plants and animals [1]. The aim of this operation is to assign a score to the degree of similarity or correlation between two sequences, for example, Protein or DNA, which can then be used to find out whether two sequences are related or not, build a multiple sequence alignment profile, or construct phylogenetic trees. The most accurate algorithms for pairwise sequence alignment are exhaustive search dynamic-programming- (DP-) based algorithms such as the Needleman-Wunsch algorithm [2] and the Smith-Waterman algorithm [3]. The latter is the most commonly used DP algorithm as it finds the best local alignment of subsegments of a pair of biological sequences. However, biological sequence alignment is also a computationally expensive application as its computing and memory requirements grow quadratically with the sequence length [4]. Given that a query sequence is often aligned to a whole database of sequences in order to find the closest matching sequence (see Figure 1) and given the annual increase in the size of biological databases, there is a need for a matching increase in computing power at reasonable cost [5].

The following subsections will present theoretical background on the Smith-Waterman algorithm, followed by an architectural overview of each of the four target hardware platforms.

2.1. The Smith-Waterman Algorithm for Pairwise Biological Sequence Alignment. Biological sequences, for example, DNA or protein sequences of residues (a DNA residue is one of four nucleotides while a protein residue is one of 20 aminoacids [1]) evolve through a process of mutation, selection, and random genetic drift [6]. Mutation, in particular, manifests itself through three main processes, namely, *substitution* of residues (i.e., a residue A in the sequence is replaced by another residue B), *insertion* of new residues, and *deletion* of existing residues. Insertion and deletion are referred to as *gaps*. The gap character "-" is introduced to present a character insertion or deletion between sequences. There are four ways to indicate the alignment between two sequence s and t as shown below:

(a, a) denotes a match (no change from s to t),

$(a, -)$ denotes deletion of character a (in s),

(a, b) denotes replacement of a (in s) by b (in t),

$(-, b)$ denotes insertion of character b (in s).

For example, an alignment of two sequences s and t (see Table 1) is an arrangement of s and t by position, where s and t can be padded with gap symbols to achieve the same length and where (A, A) indicates a match, $(G, -)$ indicates the deletion of G, $(-, T)$ indicates the insertion of T, and (C, A) indicates the replacement of C by A.

The most basic pairwise sequence analysis task is to ask whether two sequences are related or not, and by how much. It is usually done by first aligning the sequences (or part of sequences) and then deciding whether the alignment is more likely to have occurred because the sequences are related or just by chance. The parameters of the alignment methods are [1] as follows:

(i) the types of alignment to be considered;

(ii) the scoring system used to rank the alignments;

(iii) the algorithm used to find optimal (or good) scoring alignments;

(iv) the statistical methods used to evaluate the significance of an alignment score.

The degree of similarity between pairs of biological sequences is measured by a score, which is a summation of odd-log scores between pairwise residues, in addition to gap penalties. The odd-log scores are based on the statistical

Table 1: Denotations of the alignment between sequences s and t.

s:	A	G	C	A	C	A	C	$-$	C
t:	A	$-$	C	A	C	A	C	T	A

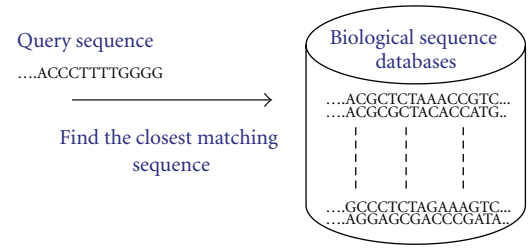

Figure 1: Pairwise sequence alignment, for example, DNA.

likelihood of any possible alignment of pairwise residues and is often summarised in a substitution matrix (e.g., BLOSUM50, BLOSUM62, PAM). The gap penalty depends on the length of gaps and is often assumed independent of the gap residues. There are two types of gap penalties, known as *linear gaps* and *affine gaps*. The linear gap is a simple model with constant gap penalty (d) multiplied by the length of the gap (g), denoted as

$$\text{Penalty}\ (g) = -g * d. \tag{1}$$

An Affine gap has opening and extension penalties. The constant penalty for opening a gap is normally bigger than the penalty for extending a gap, which is more biologically realistic as few gaps are as frequent as a single gap in practice. Affine gaps are thus formulated as (d is the opening penalty and e is the extension penalty)

$$\text{Penalty}\ (g) = -d - (g-1) * e, \quad \text{where } d > e. \tag{2}$$

For the sake of simplicity the following presents the Smith-Waterman algorithm in the case of linear gaps. The extension to the case of affine gaps is straightforward [1].

The Smith-Waterman (SW) algorithm is a widely used pairwise sequence alignment algorithm as it finds the best possible aligned subsegments in a pair of sequences (the so-called local alignment problem). It entails the construction of an alignment matrix (F) by a recursion equation as shown for an alignment between two sequences $X = \{x_i\}$ and $Y = \{y_j\}$:

$$F(i,j) = \max \begin{cases} 0, \\ F(i-1,j-1) + s\left(x_i, y_j\right), \\ F(i-1,j) - d, \\ F(i,j-1) - d. \end{cases} \tag{3}$$

Here, the alignment score is the largest of three alternatives (saturated to zero in case all three values are negative as it is better to start a new subsegment alignment than continue a subalignment with a negative score). These three alternatives are:

(i) An alignment between x_i and y_j, in which case the new score is $F(i-1, j-1) + s(x_i, y_j)$, where $s(x_i, y_j)$ is the substitution matrix score or entry for residues x_i and y_j.

(ii) An alignment between x_i and a gap in Y, in which case the new score is $F(i-1, j) - d$, where d is the gap penalty.

(iii) An alignment between y_j and a gap in X, in which case the new score is $F(i, j-1) - d$, where d is the gap penalty.

The dependency of each cell is shown in Figure 2. Here, each cell on the diagonal of the alignment matrix is independent of each other, which allows for a systolic architecture to be used in hardware in order to exploit this parallelism and hence speed up the algorithm execution.

After populating the alignment matrix, the best alignment between X and Y is obtained by tracing back from the cell with the maximum score in the alignment matrix back to the first zero matrix element. For this, we keep track of the matrix cell from which each cell's $F(i, j)$ was derived, that is, above, left, or above-left. The complete Smith-Waterman algorithm is illustrated in Figure 3 using the BLOSUM50 substitution matrix and a linear gap penalty equal to 8.

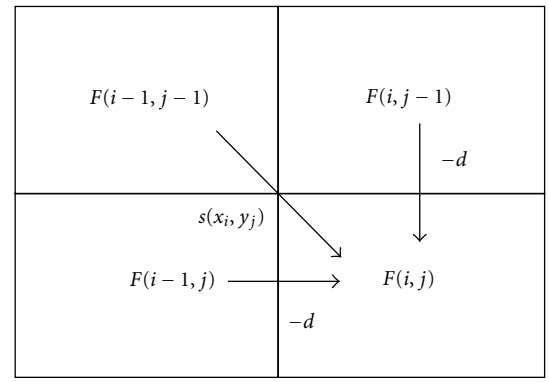

Figure 2: Data dependency of dynamic programming algorithms.

		H	E	A	G	A	W	G	H	E	E
	0	0	0	0	0	0	0	0	0	0	0
P	0	0	0	0	0	0	0	0	0	0	0
A	0	0	0	5	0	5	0	0	0	0	0
W	0	0	0	0	2	0	20	12	4	0	0
H	0	10	2	0	0	0	12	18	22	14	6
E	0	2	16	8	0	0	4	10	18	28	20
A	0	0	8	21	13	5	0	4	10	20	27
E	0	0	6	13	18	12	4	0	4	16	26

Best local alignment: A W G H E
A W-H E

Query sequence: H E A G A W G H E E

Subject sequence: P A W H E A W

Figure 3: Illustration of the Smith-Waterman algorithm.

The following subsections will present an architectural overview of each of the four implementation platforms, in turn, namely, Xilinx' Virtex-4 FPGAs, NVIDIA's GeForce 8800GTX GPU, IBM's Cell BE processor, and Intel's Pentium 4 Prescott processor. To enable a fair comparison, these specific implementation platforms were chosen because they are all based on 90 nm CMOS technology and were purchased off-the-shelf at around the same time. Moreover, each platform was targeted by a different but equally experienced programmer.

2.2. The FPGA Implementation Platform. For the purpose of our FPGA-based implementation of the Smith-Waterman algorithm, we targeted an HP ProLiant DL145 server machine [7] which has an AMD 64bit processor and a Celoxica RCHTX FPGA board [8]. The latter has a Xilinx Virtex-4 LX160-11 FPGA chip, which is based on 90 nm

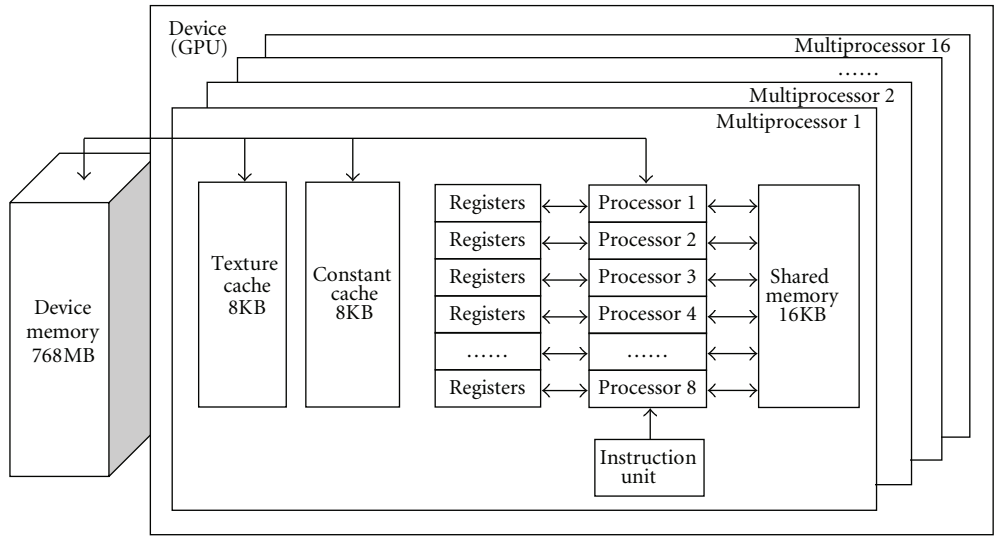

Figure 4: Architecture of NVIDIA's GeForce 8800 GTX.

copper CMOS process with a core voltage of 1.2 V [9]. All data transfer between the host processor and FPGA chip on the HP ProLiant server pass through the Hyper-Transport interface with a bandwidth of 3.2 GB/s.

The XC4VLX160 FPGA (see [9]) contains 67,584 slices, 1056 Kb of distributed memory, 96 XtremeDSP slices (not used in this paper's application) which can be configured as 18×18 multiplier with 48-bit accumulator, 288 BlockRAMs each 18 Kbit in size and configurable in dual ported mode with various word lengths and depths, and 960 user I/Os. Each slice has two 4-input look-up tables, which can be configured as 16×1 RAM, and two flip-flops in addition to some dedicated logic for fast addition and multiplication.

The FPGA design for the Smith-Waterman algorithm was captured in a C-based high level hardware language, called Handel-C [10], with the DK5 suite used to compile Handel-C into FPGA netlist, and Xilinx ISE software used for generating FPGA configuration bitstreams. A host application written in C++ services user queries and transfers them onto the FPGA board through the Hyper-Transport link. The FPGA configuration accepts a query sequence using an input/output interface based on the DSM library in Handel-C and starts alignment processing against a sequence database held on the FPGA board memory. Alignment results are then fed back to the host application through the FPGA input/output interface and Hyper-Transport link.

2.3. The GPU Platform. For the purpose of our GPU-based implementation of the Smith-Waterman algorithm, we targeted the GeForce 8800GTX GPU from NVIDIA Corp. [11]. This GPU is fabricated in 90 nm CMOS technology and consists of 16 Stream Multiprocessors (SMs), with each SM having eight Stream Processors (SPs) used as Arithmetic Logic Units (ALUs) with 8 KB constant cache, 8 KB texture cache, and 16 KB shared memory (see Figure 4). The SP clock frequency is 1,350 MHz.

This architecture, known as CUDA (Compute Unified Device Architecture), is a generic parallel computing architecture developed by NVIDIA Corp. to make the computing engines of graphics processing units accessible to general purpose software developers through a standard programming language, for example, C, with an API to exploit the architecture parallelism. Like many-core CPUs, CUDA uses threads for parallel execution. However, whereas multicore CPUs have only few threads running in parallel at any particular time, GPUs allow for thousands of parallel threads to run at the same time (768 threads per SM in the case of the GeForce 8800 GTX).

The memory hierarchy in CUDA devices consists of registers, shared memory, global memory, texture memory, and constant memory. Each SP has its own registers (1024) and operates the same kernel code as other SPs, but with different data sets. Shared memory (16 KB per SM) can be read and written to by any thread in a block of threads (or thread block) assigned to an SM. Access speed to shared memory is as fast as accessing SP registers as long as there are no bank conflicts [12]. Device memory offers global access to a larger (768 MB) but slower storage. Any thread in any SP can read from or write to any location in the global memory. Since computational results can be transferred back to CPU memory through it, global memory can be thought of as a bridge which achieves communication between GPU and CPU.

Local shared memory is allocated automatically if the size of variable required is bigger than the register size. It is not cached and cannot be accessed in a coalesced manner like global memory. Texture memory within each SM can be filled with data from the global memory. It acts as a cache, and so does constant memory, which means that their data fetch time is shorter. However, threads running in the SMs are restricted to read only access to these memories. The host CPU, on the other hand, does have write access to these memories.

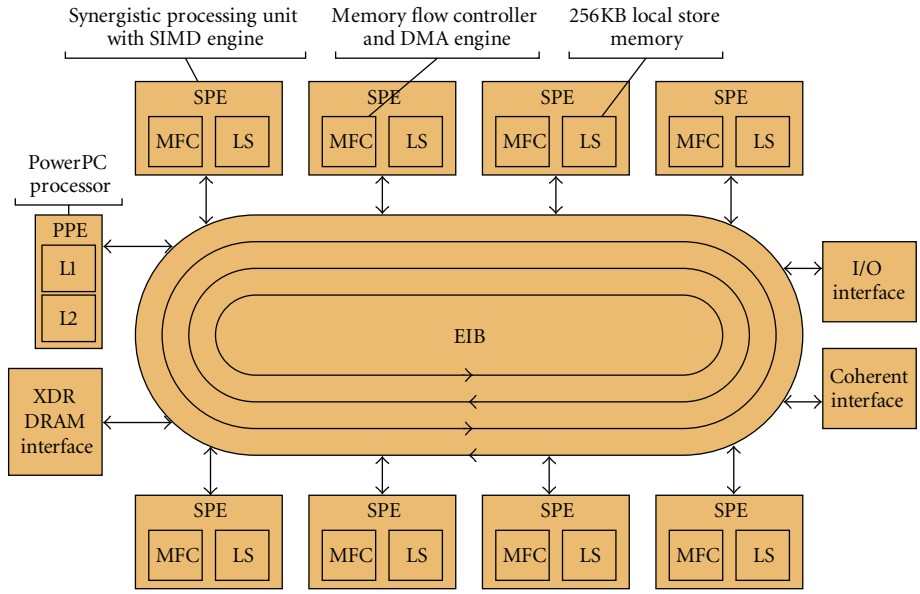

Figure 5: Architecture of Cell BE processor.

2.4. The Cell BE Platform. For the purpose of our Cell BE-based implementation, we used an IBM IntelliStation Z Pro Workstation with a Cell Acceleration Board (CAB). The CAB has one Cell Broadband Engine (Cell BE) running at 2.8 GHz, and 1 GB of XDR RAM. The IBM Cell BE is essentially a distributed memory, multiprocessing system on a single chip (see Figure 5). It consists of a ring bus that connects a single PowerPC Processing Element (PPE), eight Synergistic Processing Elements (SPEs), a high bandwidth memory interface to the external XDR main memory, and a coherent interface bus to connect multiple Cell processors together [13]. All these elements are connected with an on-chip Element Interconnect Bus (EIB). The first level instruction and data cache on the PPE are 32 KB and the level 2 cache is 512 KB. From a software perspective, the PPE can be thought of as the "host" or "control" core, where the operating system and general control functions for an application are executed.

The eight SPEs are the primary computing engines on the Cell processor. Each SPE contains a Synergistic Processing Unit (SPU), a memory flow controller, a memory management unit, a bus interface, and an atomic unit for synchronization mechanisms [14]. SPU instructions cannot access the main memory directly. Instead, they access a 256 KB local store (LS) memory, which holds both instructions and data. The programmer should keep all the codes and data within the size of LS and manage its contents by transferring data between off-chip memory and LS via mailboxes or direct memory access (DMA). This allows the programmer to overlap the computations and data transfer via double-buffering techniques [15].

We used Cell SDK version 3.0 and Mercury's MultiCore Framework (MCF) to develop our CellBE implementation. MCF uses a Function Offload Engine (FOE) model. In this model, the PPE acts as a manager directing the work of the SPEs. Sections of the algorithm in hand are loaded into the SPEs as individual "tasks." Data is then moved to the SPE where it is processed.

2.5. The GPP Platform. For the purpose of our GPP-based implementation of the Smith-Waterman algorithm, we targeted a PC with a 3.4 GHz Pentium 4 Prescott processor, 1 GB of RAM, running Windows XP OS. The Prescott processor has a 31 stage pipeline, 16 K 8-way associative L1 cache, and 1 MB L2 cache, and like all of the above platforms, it is also based on 90 nm CMOS technology.

3. Implementation of the Smith-Waterman Algorithm on FPGA

In this section, we will present the design of the Smith-Waterman algorithm implementation on FPGA. Figure 6 presents a linear systolic array for the implementation of a general purpose pairwise sequence alignment algorithm based on the dynamic programming algorithms presented in Section 2.1 above. The linear systolic array consists of a pipeline of basic processing elements (PE_i) each holding one query residue x_i, whereas the subject sequence is shifted systolically through the array [4]. Each PE performs one elementary calculation (see (3)) in one clock cycle and populates one column of the alignment matrix in turn (see Figure 7). The calculation at PE_{i+1} depends on the result from PE_i, which means that each PE is one cycle behind its predecessor. The full alignment of two sequences of lengths N and M is hence achieved in $M + N - 1$ cycles.

The architecture of Figure 6 can cater for different sequence symbol types, sequence lengths, match scores, and matching task. Indeed, the sequence symbol type, for

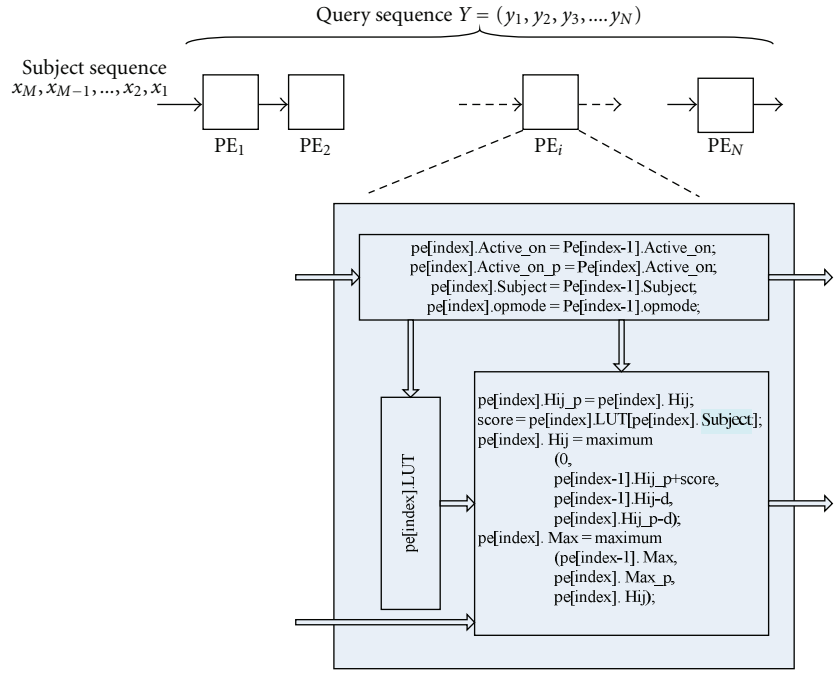

Figure 6: Linear processor array architecture for the FPGA implementation of the Smith-Waterman algorithm.

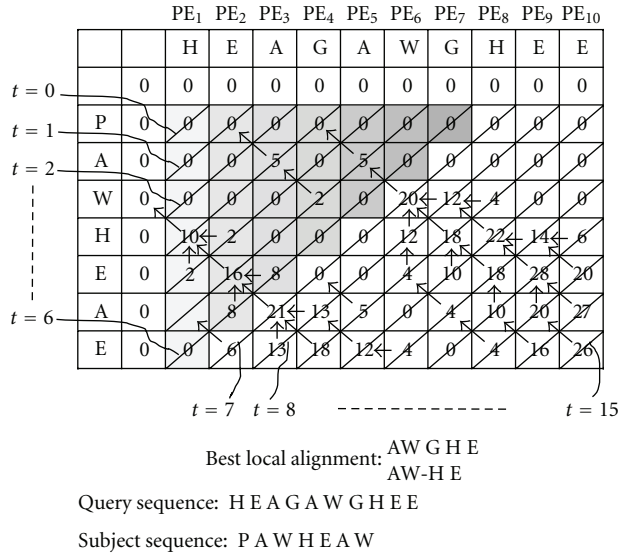

Figure 7: Illustration of the execution of the Smith-Waterman on the linear array processor.

example, DNA or Proteins, will only influence the word length of the input sequence, for example, 2 bits for DNA and 5 bits for Proteins, the query sequence length dictates the number of PEs, and the match score attributed to a symbol match depends on the substitution matrix used. Given a particular substitution matrix, for example, BLOSUM50, all possible match scores for a particular symbol represent one column in the substitution matrix. Hence, for each PE, we store the corresponding symbol's column in the substitution matrix, which we use as a look-up table. A different substitution matrix will hence simply mean a different look-up table content. The penalties attributed to a gap can also be stored in the PE.

The linear array of Figure 6 can also cater for different matching tasks with few changes. For instance, the difference between global alignment, local alignment, and overlapped matching [1, 4] resides in the initial values of the alignment matrix (border values), the recursive equation implemented by the PE, as well as the starting cell of the traceback procedure. Although a query sequence is often compared to a large set of database sequences, the traceback procedure is only needed for few sequences with high alignment scores. As such, it is customary to perform this on a host (sequential) processor as the time involved in this operation is negligible compared to the time it takes to align the query sequence against a whole sequence database.

3.1. The Case of Long Sequences. The number of PEs that could be implemented on an FPGA is limited by the logic resources available on-chip. For instance, the maximum number of PEs that could be implemented on a Xilinx XC2V6000 Virtex-II FPGA in the case of the Smith-Waterman algorithm with affine gap penalties is ~250. Clearly, this is not sufficient for many real world sequences where query sequence lengths can be in the thousands. The solution in such cases is to partition the algorithm in hand into small alignment steps and map the partitioned algorithm onto a fixed size linear systolic array (whose size is dictated by the FPGA in hand) as illustrated in Figure 8 below [4]. Here, the sequence alignment is performed in several passes. A First-In-First-Out (FIFO) memory block

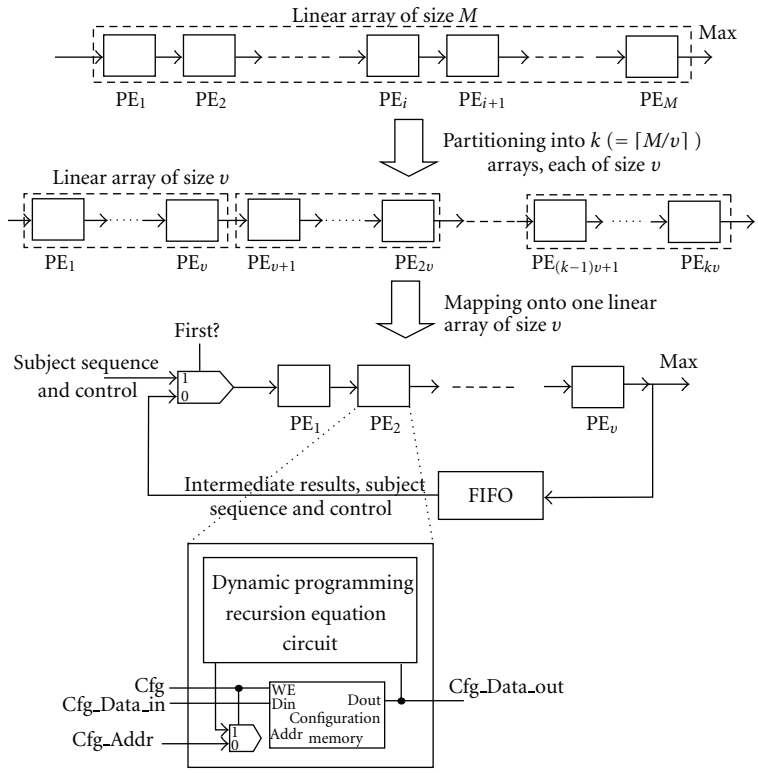

Figure 8: Partitioning/Mapping of a sequence alignment algorithm onto a fixed size systolic array.

is used to store intermediate results from each pass before they are fed back to the array input for the next pass. The depth of the FIFO is dictated by the length of the subject sequence. Another consequence of the folded architecture is that each PE should now hold k substitution matrix columns (or look-up tables) instead of just one. In order to load the initial values of the look-up tables used by the PEs, a serial configuration chain is used, as illustrated in Figure 8. When the control bit *Cfg* is set to 1, the circuit is in configuration mode. Distributed memory in each PE then behaves as a register chain. Each PE configuration memory is loaded with the corresponding look-up tables sequentially. At the end of the configuration, *Cfg* is reset to 0 indicating the start of the operation mode.

4. Implementation of the Smith-Waterman Algorithm on GPU

The parallelization strategy adopted for our GPU implementation is based on multithreading and multiprocessing. Indeed, several threads are allocated to the computation of a single alignment matrix in parallel within a thread block, while several thread blocks are allocated to compute the alignments of different pairs of sequences [16]. We separate a single alignment matrix computation into multiple submatrices with a certain number of threads allocated to calculate each submatrix in parallel, depending on the maximum number of threads and maximum amount of memory available (see Figure 9). Once the allocated batch of threads completes a sub-matrix calculation, the final thread in the batch records the data in the row of which it takes charge and stores it into shared memory or global memory, depending on the size of database subject sequence, ready for the calculation of the next sub-matrix. The first thread in the batch, on the other hand, loads this data as initial data for the subsequent sub-matrix calculation. This operation continues in turn until the end of the entire alignment matrix calculation.

The above procedure makes this GPU implementation scalable to any sequence length. On the NVIDIA GeForce 8800GTX GPU, each SM can have 768 parallel threads running at the same time. Hence, we split this number into batches of threads or blocks, where each block computes one alignment matrix. For example, we can split the overall number of threads into 8 blocks of 96 threads, with 10 registers allocated to each thread and each block could use almost 2 KB of shared memory. Global memory will be used if this amount of allocated shared memory space is not enough for any database subject sequence. Note here that if the length of the database subject sequence is smaller than the number of threads in the block, additional waiting time should be added for the threads in the batch to finish their computations. This is easy to imagine, for example, if thread 0 has already completed its row calculation, but thread n has not completed yet or has not even started its row, then thread 0 would have to wait for thread n of the previous sub-matrix alignment to complete its task before obtaining its initial data for the sub-matrix alignment.

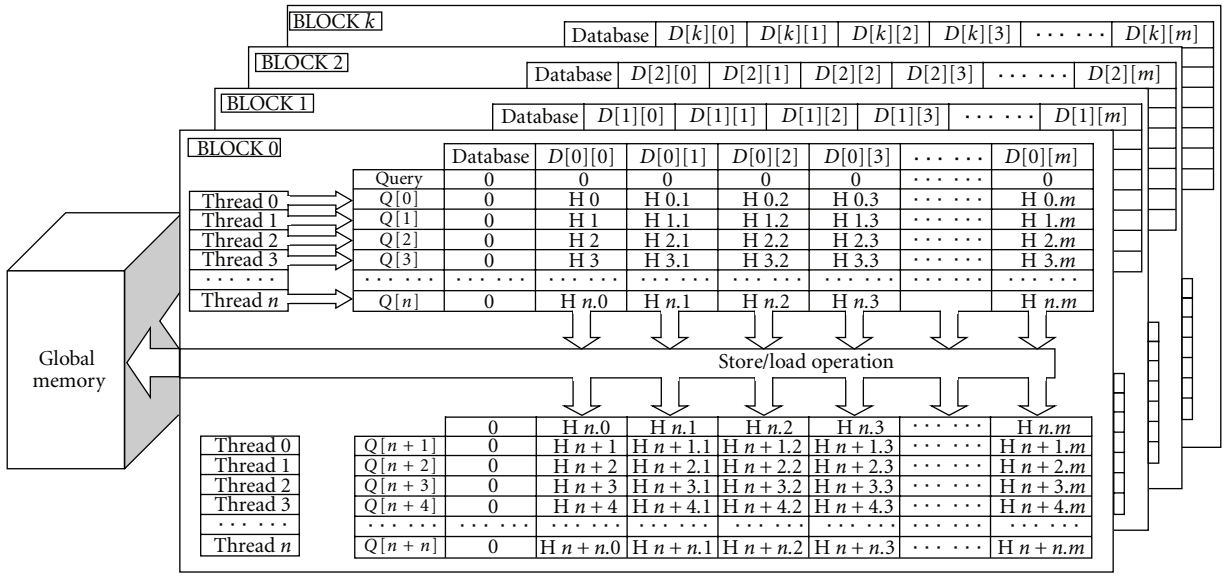

Figure 9: Our GPU parallel thread implementation of the Smith-Waterman algorithm: store and load operations are performed by the final thread and the first thread in each thread batch (block) to allow for any sequence length processing.

Thread 0	$Q[0]$	$D[0], D[1]$	$D[2], D[3]$	$D[4], D[5]$	⋯ ⋯	$D[m-1], D[m]$
Thread 1	$Q[1]$	$D[0], D[1]$	$D[2], D[3]$	$D[4], D[5]$	⋯ ⋯	$D[m-1], D[m]$
Thread 2	$Q[2]$	$D[0], D[1]$	$D[2], D[3]$	$D[4], D[5]$	⋯ ⋯	$D[m-1], D[m]$
⋯ ⋯	⋯ ⋯	⋯ ⋯	⋯ ⋯	⋯ ⋯	⋯ ⋯	$D[m-1], D[m]$
Thread n	$Q[n]$	$D[0], D[1]$	$D[2], D[3]$	$D[4], D[5]$	⋯ ⋯	$D[m-1], D[m]$

Figure 10: Alignment matrix calculation with vector variable char2.

4.1. Load Partitioning Strategy. In our GPU implementation, we used constant cache to store the commonly-used constant parameters in order to decrease access time, including the substitution matrix and the query sequence. In addition, we used global memory to store the database sequence as the size of the latter can be in the hundreds of Megabytes. Moreover, we used texture cache to shade database sequences. The bottleneck of speed-up in our implementation is the store operation of temporary data by the last thread and the load operation by the first thread in each batch, because the latency between SP registers and global memory is much longer than the one between registers and shared memory. No matter how fast other threads can execute the kernel code, they have to wait for a synchronization point of all threads. Obviously, this only occurs when the length of the database subject sequence is longer than the allocated space in shared memory. Therefore, our acceleration strategy mainly focuses on the efficient allocation of resources to each block to make the maximum use of the available parallelism. This can be achieved through setting the proper number of threads in each block. Here, since each SM has 8192 registers and can keep at most 768 parallel threads, for a query sequence of length 512, if we use 1 block of 512 threads, 16 registers can be used for each thread. In this case, only one pairwise sequence alignment can be computed in each SM. If we use 8 blocks of 64 threads, also 16 registers can be allocated to each thread, but the number of sequence alignments that can be processed at the same time becomes 8. Rather than adopting the simple method used in [17] which utilizes the full memory resources for each block, we flexibly allocate resources through setting the number of threads in each block, with no limitation on the overall length of the query sequence. Table 2 presents execution times of the Smith-Waterman algorithm on GPU using our technique, with different numbers of threads per block. For a query sequence of length 1023, 64 threads per block leads to the best performance.

Note finally that we used vector type *char2* as illustrated in Figure 10 to decrease the data fetch times compared to using *char* [12]. This was empirically found to be more efficient than using vector *char4*.

5. Implementation of the Smith-Waterman on IBM Cell BE

Our IBM Cell BE implementation exploits the data parallelism involved in aligning one query sequence with a large database of sequences by assigning different database

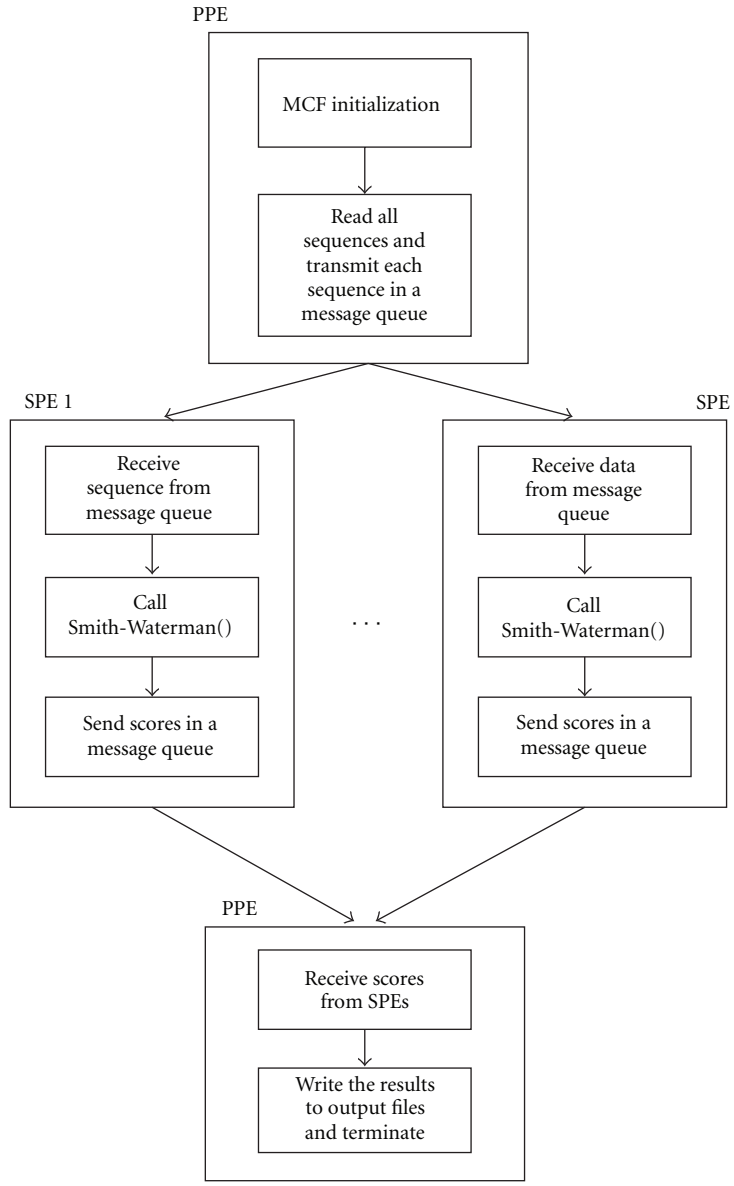

Figure 11: The parallel code flow of Smith-Waterman algorithm on Cell BE. The PPE transmits data to SPEs via one message queue and receives the results from SPEs by another.

sequences to the eight SPEs, that is, through multiprocessing. The PPE operates as a manager handling data I/O, assigning tasks and scheduling the SPEs. It reads input database sequences from disk, transmits different database sequences to different SPEs, and invokes SPEs to perform pairwise sequence alignment using the Smith-Waterman algorithm on independent sequences as illustrated in Figure 11.

The Cell BE uses two schemes to transfer data between the DRAM and the SPEs: (1) message queue and (2) direct memory access (DMA). In the message queue mode, the PPE reads the data from the DRAM, packetizes the data, and places packets into the message queue for the target SPE to read. DMA is a mechanism for an SPE to bypass the PPE and read data from the DRAM directly. This feature makes DMA a desirable option for data intensive applications. However, based on our timing trace results on the Cell BE, we found that the computation time within each SPE is the dominant component of the total execution time. For the case of query sequence length 256, we observed that 92.7% of the time is spent on computation and only 7.3% on the data transfer with the message queue. From this perspective, switching from message queue to DMA will not improve the performance considerably. Indeed, the average sequence length of the SWISS-PROT database is about 360, which can be completely packetized into a message queue and transmitted between PPE and SPEs. Therefore, we chose message queue as the parallelism strategy on the Cell BE due to the short bandwidth and latency of data communication [19].

Table 2: Performance comparison for different numbers of threads per block (64, 128, 256). All query sequences run against the SWISS-PROT database [18].

Query length	Thread 64 time (sec)	Thread 128 time (sec)	Thread 256 time (sec)
63	2.1	3.1	6.2
127	6.1	4.2	7.2
191	9.3	11.9	8.3
255	12.5	12.9	9.6
511	25.1	26.4	29.2
1023	50.4	53.1	57.8

We packetize and transmit each sequence in a message queue between PPE and SPEs. First, the manager and the workers all initialize the MCF network. Then, the PPE feeds the worker code into the local store memory of each SPE and signals them to start. As part of the initialization process, we dynamically set up two message queues, one is for PPE sending data to SPEs, and the other is backwards, for SPEs passing results back towards the PPE. After reading one sequence from the database, the manager puts it into one message queue and sets up a loop in which the PPE sends the message to SPEs separately. The manager then waits for the synchronization semaphore from the SPEs when they finish pulling the data into local store. Sequentially, the SPEs start processing the data in a concurrent manner. Whoever completes its computation first sends the results back to the PPE by means of the other message queue. This process continues until PPE transmits all the sequences to the SPEs. The manager then deallocates memory, destroys the MCF network, and terminates the program.

Parallelization of any algorithm requires careful attention to the partitioning of tasks and data across the system resources. Each SPE has a 256 KB local store memory that is allocated between executable code and data. The executable code section must contain the top-level worker code, the MCF functions, and any additional library functions that are used. If the total amount of executable code is too large for the allocated memory, it may be loaded as several "plug-ins." If the total amount of data exceeds the data allocation, it may be loaded down as "tiles." MCF contains plug-in and tile channel constructs to facilitate this as required. The tradeoff here is in increased code complexity. The core functions of the Smith-Waterman algorithm implementation compile to less than 83 KB. MCF adds up to 64 KB depending on the functions that are used. Rounding this up suggests that the worker code would somewhat be greater than half of the available SPE memory (128 KB). For our specific database, the maximum length of all the sequences is 35,213 bytes, which amounts to ~36 KB of data. These estimates suggest that each SPE could receive a full code segment and a complete set of protein sequence without the need for further partitioning.

Inside each SPE, a pairwise sequence alignment using the Smith-Waterman algorithm is performed column-wise, four cells at a time as illustrated in Figure 12 for a database sequence of length 4 and a query of length 8. When cells are calculated, we keep track of their updated values in a temporary register (cell calculations) which is updated each time a new column is calculated. The entire pairwise alignment matrix is not stored in memory, but rather just the temporary cell calculations column. Four dependency values are read at the beginning of an inner loop, and the new values for which the next column will be dependent are written at the end of the inner loop. The number of dependent cells needed for each alignment is simply equal to the length of the query sequence, since we are calculating cells column by column. After all SPE pairwise alignments are completed, the highest pairwise score calculated by each SPE is returned to the main program (in the PPE) for final reduction. The sequence with the highest score achieves the best alignment. Finally, it is worth mentioning that, currently, our query lengths are limited to 1024 residues, but we are working on some indexing strategies which will allow us to increase the length of a query.

6. Implementation of the Smith-Waterman Algorithm on GPP

Since our aim is to compare all four technologies not just in terms of speed, but also in terms of energy consumption, and purchase and development costs, we chose to use a widely adopted GPP implementation of the Smith-Waterman algorithm, namely, SSEARCH (version 35.04) from the FASTA set of programs [20]. SSEARCH was run on a 3.4 GHz Pentium 4 Prescott processor with 1 GB RAM, running Windows XP Professional. Using the SSEARCH program is perfectly justified for the purpose of this paper since it is a mature piece of software that is widely used by Bioinformaticians in practice. We are aware of better GPP implementations in the literature, for example, [21]. However, here again, such implementations do not give us the development time, for instance, nor do they guarantee a fair balance of experience between the developers of each implementation. They hence do not serve the particular aims of this paper.

7. Comparative Implementation Results and Evaluation

This section presents the implementation results of our Smith-Waterman designs on FPGA, GPU, Cell BE, and GPP. Table 3 first presents the execution times of the Smith-Waterman implementation on all four platforms for a number of query sequences against the SWISS-PROT database (as of August 2008) when it contained 392,768 sequences and a total of 141,218,456 characters. For the FPGA, GPU, and Cell BE implementations, we assume that the database is already on the accelerator card's memory. Thus, the execution times shown in Table 3 do not include the database transfer time as it is an initial step. In practice, queries are made against fairly static databases, and hence this assumption is reasonable.

Note that for smaller sequences, the target FPGA chip could easily fit more processing elements on chip and thus

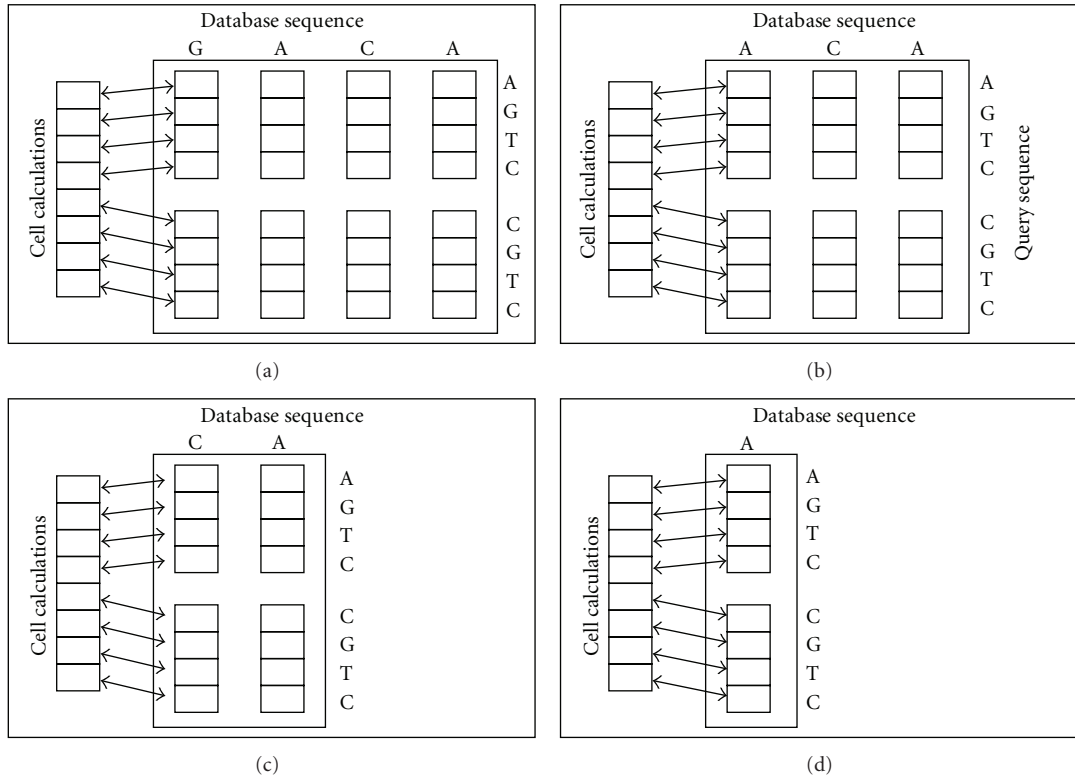

Figure 12: Illustration of the Smith-Waterman calculation on the Cell BE.

Table 3: Performance comparison. All query sequences run against the SWISS-PROT database.

Query (protein name)	Query length	FPGA time (sec)	GPU time (sec)	Cell BE time (sec)	GPP time (sec)
P36515	4	1.5	4.1	0.5	24
P81780	8	1.6	4.1	1.0	30
P83511	16	1.6	4.3	1.3	43
O19927	32	1.6	4.7	1.4	62
A4T9V0	64	1.6	6.7	2.5	115
Q2IJ63	128	1.6	12.8	5.1	210
P28484	256	1.9	30.0	9.4	424
Q1JLB7	512	4.5	76	17.2	779
A2Q8L1	768	6.7	136.2	22.2	1356
P08715	1024	8.9	172.8	31.8	1817

(provided there is enough bandwidth to transfer more data to the FPGA chip in parallel) the execution time could be reduced several fold. A fairer comparison in speed would take the results of sequence length 256 since it is close to the average sequence length in the SWISS-PROT database (360). Each PE in the FPGA systolic array consumes ~110 slices, and, consequently, we were able to fit ~500 PEs on a Xilinx Virtex-4 LX160-11 FPGA [22]. Moreover, the processing word length in the FPGA systolic array is 16 bits, and the circuit was clocked at 80 MHz. Table 4 presents the corresponding performance figures in Giga Cell Updates Per Seconds (GCUPS) (the CUPS (or Cell Updates Per Second) is a common performance measure used in computational biology. Its inverse represents the equivalent time needed for a complete computation of one entry of the alignment matrix) as well as the speed-up figures normalized with respect to the GPP implementation result. This shows the FPGA solution to be two orders of magnitude quicker than the GPP solution, with the Cell BE and GPU coming second and third, respectively. The latter two achieve one order of magnitude speed-up compared to GPP.

We note before embarking on result evaluation that faster implementations do exist in the literature. For instance, in [21], the author presented a GPP implementation of the Smith-Waterman algorithm on a 2.0 GHz Xeon Core 2 Duo processor with 2 GB of RAM running Windows

Table 4: Performance comparison for query sequence of length 256.

Platform	GCUPS	Speed-up
FPGA	19.4	228 : 1
GPU	1.2	14 : 1
Cell BE	3.84	45 : 1
GPP	0.085	1 : 1

Table 5: Development times of the Smith-Waterman algorithm implementation on all four technologies.

Platform	Development time in days
FPGA	300
GPU	45
Cell BE	90
GPP	1

XP SP2. The software implementation exploited Intel SSE2 technology and resulted in a much higher performance of 1.37 GCUPS. Moreover, a Smith-Waterman implementation on an NVIDIA GTX 295 Dual Core GPU, which contains 30 SPs, 896 MB memory per GPU core, installed on a PC with a Core 2 Duo E7200 2.53 GHz processor, with 2 GB RAM, and running Cent OS 5.0, resulted in ∼11 GCUPS performance [23]. This shows that GPPs and GPUs can outperform the above results considerably with more design effort, for example, exploiting SSE2 technology in [21], and optimizing thread scheduling and memory architecture [23] as well as exploiting more advanced process technologies (below 90 nm). However, the aim of this paper is to perform a fair comparison of all four technologies in terms of speed, development time, cost, and energy consumption. For instance, the process technology of the GPP and GPU devices reported in [21, 23], respectively were more advanced than the FPGA technology we used in this study. Moreover, these implementations do not report the development time which is crucial to assess productivity as will be shown below. As such, the following will concentrate on the results shown in Table 4 above rather than other implementations reported in the literature as these do not serve the particular aims of this paper, despite their worth.

In order to put the speed-up figures shown in Table 4 into perspective, we measured the time it took to develop each of these implementations. Indeed, each of the four implementations was developed by a different Ph.D. student with a comparable experience in programming his/her respective platform. Table 5 presents the resulting development times.

This shows FPGA development time to be one order of magnitude higher than that of Cell BE and GPU, and two orders of magnitude higher than that of GPP. It is worth mentioning that the majority of FPGA development time (∼80%) was spent in learning the specific FPGA hardware application programming interface (API) as well as debugging the FPGA implementation in hardware. As such the choice of the hardware description language (e.g., VHDL or Verilog instead of Handel-C) in itself would not have had a major impact on the figures. The lack of standards (e.g., standard FPGA hardware boards, standard FPGA APIs) however remains a major problem for FPGA programmers.

Table 6: Cost of purchase and development of the Smith-Waterman algorithm implementation on all four technologies.

Platform	Purchase cost ($)	Development cost ($)	Overall cost ($)	Normalized overall cost
FPGA	10,000	48,000	58,000	50
GPU	1450	7,200	8,650	8
Cell BE	8,000	14,400	22,400	19
GPP	1000	160	1160	1

Table 7: Performance per $ spent for each technology.

Platform	Performance (MCUPS) per $ spent	Normalized performance per $ spent
FPGA	0.34	4.6
GPU	0.14	1.9
Cell BE	0.17	2.3
GPP	0.07	1

By accounting for the cost of development (measured on the basis of US$20/hour as the average salary of a freshly graduated student where the experiments took place) and the cost of purchase of the respective platforms, Table 6 gives the overall development cost of all four solutions. Note here that the purchase cost of FPGA, GPU and Cell BE includes the cost of the host machine.

We can see that the FPGA solution is 50x more expensive than the GPP solution, followed by the Cell BE (19x) and the GPU (8x). Based on these figures, we can measure the performance per dollar spent by dividing the GCUPS figures of Table 4 by the overall cost figures given in Table 6 for each platform. The results are presented in Table 7 below (performance is expressed in Mega CUPS per dollar).

This shows the FPGA platform to be a more economical solution for this particular algorithm despite its relatively high cost, thanks to its much higher performance. The CellBE and GPU came second and third, respectively.

We have also measured the power consumed by each implementation (excluding the host in the case of FPGA, GPU, and Cell BE) as shown in Table 8. We used a power meter connected between the power socket and the machine under test for this purpose. We noted the power meter reading, at steady state, when the Smith-Waterman algorithm was running. This includes two parts: an idle power component and a dynamic power component. The idle power component can be obtained from the power meter when the machine is in the idle state, which means that no Smith-Waterman algorithm implementation was running on it. The dynamic power consumption is thus obtained by deducting the idle power reading from the steady state power reading. The power measurement results are shown in Table 8.

Table 8: Power consumption of the Smith-Waterman algorithm implementation on all four technologies.

Platform	Idle power (watt)	Steady state power (watt)
FPGA (clocked at 80 MHz)	100	139
GPU	70	126
Cell BE	180	140
GPP	70	100

Table 9: Power and energy consumption of the Smith-Waterman algorithm implementation on all four technologies.

Platform	Power (watt)	Energy (joule)	Normalized energy consumption
FPGA (clocked at 80 MHz)	39	73	0.0017
GPU	56	1682	0.04
Cell BE	140	1317	0.03
GPP	100	42400	1

Table 10: Performance per watt figures of the Smith-Waterman algorithm implementation on all four technologies.

Platform	Performance (MCUPS) per watt	Normalized performance per watt
FPGA	508	584
GPU	22	25
Cell BE	27	31
GPP	0.87	1

We use the dynamic power figures for the accelerated implementations, that is, the FPGA, GPU, and Cell BE-based implementations, as nearly all of the processing is done on the accelerator, with the host only sending query data and collecting results from the accelerator. As such, the cost and power consumption of the host could be made as small as needed without affecting the overall solution performance. The GPP implementation's steady state power figure however is used, instead of the dynamic power, as there is no distinction between host and accelerator in this case.

Multiplying the power figure for each platform with the execution time, we obtain the energy consumed by each implementation as shown in the Table 9.

This shows the FPGA solution to be three orders of magnitude more energy efficient than GPP, while the Cell BE and the GPU came second and third, respectively (with one order of magnitude energy efficiency compared to GPP). The performance per watt figure can thus be calculated by dividing the GCUPS figures of Table 4 by the power consumption figure in Table 9 for each platform. The results are presented in Table 10 (performance is expressed in Mega CUPS per watt).

This again highlights the high energy efficiency of the FPGA solution, followed by Cell BE and GPU. The latter is often unfairly characterized as energy inefficient in the computing community, something that the results of this study dispute. Indeed, factoring the speed-up gains, GPUs can be much more energy efficient than GPPs, as shown in this study.

Table 11: Performance per $ and per watt for each technology using the GPP implementation of [21] and GPU implementation of [23].

Platform	Performance (MCUPS) per $	Performance (MCUPS) per watt
FPGA	0.34	508
GPU	1.27	196
Cell BE	0.17	27
GPP	1.18	13.7

It is important to note at this stage that the above results are very sensitive to the technology used and level of effort spent on the implementation. For instance, if we consider the GPP and GPU implementations reported in [21, 23], respectively, and assuming that development times and power consumption figures were similar to the GPP and GPU implementations reported in this paper, then the resulting performance per $ and performance per watt figures of the GPP and GPU implementations would have been as shown in Table 11.

This shows GPUs to be more economic on performance per $ grounds compared with other technologies, followed closely by GPPs. Such conclusion however is not valid according to the criteria set in this paper, since the GPP and GPU implementations reported in [21, 23], respectively were based on more advanced process technologies, brought to market after Virtex-4 FPGAs. In addition, these implementations needed more design effort. Nonetheless, we note that the performance per watt figures in Table 11 still show FPGAs to be far superior to the other technologies on energy efficiency criterion.

We note finally that the Smith-Waterman algorithm implementation scales extremely well with data sizes and computing resources with the four technologies used (FPGAs, GPUs, Cell BE, or GPP). Indeed, the algorithm is characterized by high level data and instruction level parallelism, and given the parameterizable way in which we designed our solutions, the same piece of code can be used to take advantage of increasing resources on the target platform, for example, FPGAs with more slices and memory, GPUs with more stream processors and threads, Cell BEs with more SPEs. In the case of GPUs, for instance, allocating different pairwise alignments to extra GPU stream processors would increase the speed-up proportionally, assuming memory bandwidth is also increased. Beyond a single chip, a straightforward way of scaling up the algorithm is to split the subject database into N subdatabases and allocate each subdatabase to one GPU chip. Partial results are then reduced by a host processor in a typical scatter-gather manner, as demonstrated in [23]. The same reasoning applies to general purpose processors and Cell BE. As for FPGAs, bigger chips would result in an increase of the number of PEs that could fit on chip, which would in turn increase

the GCUPS performance proportionally, provided proper control circuitry is employed to use all PEs (or a large proportion of them) at any given time. Such techniques were illustrated in [5]. In general, and given the high computation to communication ratio of the algorithm, the scalability of the execution time as a function of available hardware resources is near linear.

In view of the above, the following conclusion section summarizes the findings of this study and presents a number of general lessons learnt through it.

8. Conclusion

This paper showed the design and parallel implementation of the Smith-Waterman algorithm on three different technologies: FPGA, GPU, and IBM Cell BE and compared the results with a standard sequential GPP implementation. Comparison criteria included speed, development time, overall cost, and energy consumption. This study revealed FPGAs to be a cost effective and energy efficient platform for the implementation of the Smith-Waterman algorithm as it came on top on both performance per dollar and performance per watt criteria. FPGAs achieved 4.6x more performance for each $ spent and 584x more performance for each Watt consumed, compared to GPPs. The IBM Cell BE came second as it achieved 2.3x more performance for each $ spent and 31x more performance for each Watt consumed, compared to CPUs. Finally, the GPU came third as it achieved 1.9x more performance for each $ spent and 25x more performance for each watt consumed, compared to GPPs.

The speed of FPGA implementation was limited by the amount of logic resources on chip as more parallelism could be obtained with more processing elements. As for the GPU, the parallelism was limited by the size of local memory (shared memory and number of registers) as well as the number of parallel processes and threads that could be launched at the same time, which is dictated by the number of stream processors and their parallelism potential. Finally, for the Cell-BE, the parallelism was mainly limited by the number of Synergistic Processing Elements (SPEs) and the parallelism potential of each SPE. Specifically, with the Cell BE, it is not feasible to match the fine grained parallelism level of the GPU. Indeed in the Cell BE implementation, we divided the workload equally among the SPEs and let each SPE run the sequence alignment algorithm on its own data set. If an SPE is assigned to process "n" sequences, the program is executed over these one after the other, in a sequential manner.

We note, however, that the overall cost of the implementations did not include the energy cost as this would depend on the amount of use of the platform. More importantly perhaps these calculations did not account for issues such as technology maturity, backward and forward compatibility, and algorithms' rate of change, which all play an important role in technology procurement decisions. Unfortunately, these issues are more difficult to quantify and are often subjective and time/location-sensitive. We also note that the comparison presented in this paper has been conducted for one single algorithm, which limits the generalizability of the results. Indeed, we recognize that our experiment is not statistically significant and that development times for instance would vary significantly for one programmer to another. Nonetheless, our objective was to put the "speed-up" values achieved into perspective with an analysis of productivity based on the personal experience of an average Ph.D. student. As such, the following general lessons could be learnt from the case study presented in this paper.

First, the reason FPGAs outperform other architectures in this Smith-Waterman case study is three-fold: (1) the high level of data and instruction level parallelism available in the algorithm which suits FPGA distributed resources very well, (2) the fine granularity of the instructions involved which suits the fine-grained FPGA computing resources (e.g., abundant 4-bit lookup tables and shift registers), and (3) the relatively low dynamic range requirement, which means that fixed point arithmetic with a relatively small number of bits can be used, which also suits FPGAs' fine-grained architectures. Thus, any application with high fine-grained instruction and data level parallelism, and modest dynamic range data requirements should be expected to achieve similar performance gains on FPGAs. General purpose processors should be expected to fare better for less parallel/regular algorithms on the other hand. Second, FPGA technology's main competitive advantage is on performance per watt criteria. High performance computing applications where power consumption is often a bottleneck should hence benefit from this technology. Third, on the economic viability front (i.e., performance per dollar spent), and using currently available hardware and development tools, FPGAs need to achieve at least two orders of magnitude speed-up compared to GPPs and one order of magnitude speed-up compared to GPUs and CellBE to justify their relatively longer development times and higher purchase costs. This relatively high hurdle represents a major problem in the face of further market penetration for FPGA technology, and it is mainly due to FPGAs' relatively longer development times. Standard FPGA boards with standard communication and application programming interfaces can lower the aforementioned hurdle drastically as the majority of FPGA development time is often spent on learning and debugging specific FPGA hardware application programming interfaces and tools.

References

[1] R. Durbin, S. Eddy, S. Krogh, and G. Michison, *Biological Sequence Analysis: Probabilistic Models for Proteins and Nucleic Acids*, Cambridge University Press, 1998.

[2] S. B. Needleman and C. D. Wunsch, "A general method applicable to the search for similarities in the amino acid sequence of two proteins," *Journal of Molecular Biology*, vol. 48, no. 3, pp. 443–453, 1970.

[3] T. F. Smith and M. S. Waterman, "Identification of common molecular subsequences," *Journal of Molecular Biology*, vol. 147, no. 1, pp. 195–197, 1981.

[4] K. Benkrid, Y. Liu, and A. Benkrid, "A highly parameterized and efficient FPGA-Based skeleton for pairwise biological

sequence alignment," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 17, no. 4, pp. 561–570, 2009.

[5] T. Oliver, B. Schmidt, and D. Maskell, "Hyper customized processors for bio-sequence database scanning on FPGAs," in *Proceedings of the ACM/SIGDA 13th ACM International Symposium on Field Programmable Gate Arrays (FPGA '05)*, pp. 229–237, February 2005.

[6] G. A. Harrison, J. M. Tanner, D. R. Pilbeam, and P. T. Baker, *Human Biology: An Introduction to Human Evolution, Variation, Growth, and Adaptability*, Oxford Science, 1988.

[7] HP Proliant DL145 Server Series, Hewlett-Packard, 2007, http://www.hp.com/.

[8] Celoxica, The RCHTX FPGA Acceleration Card Data Sheets, Celoxica, 2007, http://www.hypertransport.org/docs/tech/rchtx_datasheet_screen.pdf.

[9] Xilinx Corporation, Virtex-4 Family Data Sheets, 2007, http://www.xilinx.com/support/documentation/virtex-4_data_sheets.htm.

[10] Agility DS, Handel-C Reference Manual, 2009, http://www.mentor.com/products/fpga/handel-c/upload/handelc-reference.pdf.

[11] Nvidia Corporation, GeForce 8800 GPUs, 2009, http://www.nvidia.co.uk/page/geforce_8800.html.

[12] CUDA, CUDA Programming Guide Version 1.1, NVIDIA Corporation, 2009, http://developer.nvidia.com/cuda/.

[13] Mercury Computer Systems, Datasheet: Cell Accelerator Board, 2009, http://www.mc.com/.

[14] F. Blagojevic, D. S. Nikolopous, A. Stamatkis, and C. D. Antonopoulos, "Dynamic multigrain parallelization on the cell broadband engine," in *Proceedings of the ACM SIGPLAN Principles and Practice of Parallel Computing (PPoPP '07)*, San Jose, Calif, USA, March 2007.

[15] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy, "Introduction to the cell multiprocessor," *IBM Journal of Research and Development*, vol. 49, no. 4-5, pp. 589–604, 2005.

[16] C. Ling, K. Benkrid, and T. Hamada, "A parameterisable and scalable smith-Waterman algorithm implementation on CUDA-compatible GPUs," in *Proceedings of the IEEE 7th Symposium on Application Specific Processors (SASP '09)*, pp. 94–100, July 2009.

[17] Y. Munekawa, F. Ino, and K. Hagihara, Design and Implementation of the Smith-Waterman Algorithm on the CUDA-Compatible GPU, 2008.

[18] A. Bairoch and R. Apweiler, The SWISS-PROT protein knowledgebase and its supplement TrEMBL, *Nucleic Acid Research, Release* 56.3, October 2008.

[19] Y. Song, G. M. Striemer, and A. Akoglu, "Performance analysis of IBM Cell Broadband Engine on sequence alignment," in *Proceedings of the NASA/ESA Conference on Adaptive Hardware and Systems (AHS '09)*, pp. 439–446, August 2009.

[20] W. R. Pearson and D. J. Lipman, "Improved tools for biological sequence comparison," *Proceedings of the National Academy of Sciences of the United States of America*, vol. 85, no. 8, pp. 2444–2448, 1988.

[21] M. Farrar, "Striped Smith-Waterman speeds database searches six times over other SIMD implementations," *Bioinformatics*, vol. 23, no. 2, pp. 156–161, 2007.

[22] Y. Liu, K. Benkrid, A. Benkrid, and S. Kasap, "An FPGA-based web server for high performance biological sequence alignment," in *Proceedings of the NASA/ESA Conference on Adaptive Hardware and Systems (AHS '09)*, pp. 361–368, August 2009.

[23] K. Dohi, K. Benkrid, C. Ling, T. Hamada, and Y. Shibata, "Highly efficient mapping of the Smith-Waterman algorithm on CUDA-compatible GPUs," in *Proceedings of the 21st IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP '10)*, pp. 29–36, July 2010.